AD-A267 153

ENTATION PAGE

Form Approved
OMB No 0704-0188

ORT DATE	3. REPORT TYPE AND DATES COVERED
	FINAL/01 APR 90 TO 30 SEP 92

4. TITLE AND SUBTITLE
MONOTONE APPROXIMATE QUERY PROCESSING (U)

5. FUNDING NUMBERS
2304/A7
AFOSR-90-0193

6. AUTHOR(S)
Professor Jane Liu

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Illinois
Dept of Computer Science
Urbana IL 61801-2987

8. PERFORMING ORGANIZATION REPORT NUMBER
AFOSR-TR- 93 0478

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFOSR/NM
110 DUNCAN AVE, SUTE B115
BOLLING AFB DC 20332-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFOSR-90-0193

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE: DISTRIBUTION IS UNLIMITED

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This project was concerned with means to provide approximate answers to queries. The researchers developed a monotone query processing scheme. This scheme allows a data base system to provide for each a series of intermediate answers that are approximations of the exact answer. The approximate answers improve monotonically in accuracy as more data are retrieved and processed to answer the query. If for any reason query processing must be terminated prematurely before the exact answer is produced, the latest intermediate answer, that is, the best approximate answer produced so far, is made available to the application. For many time-critical applications, a timely approximate answer that is sufficiently good is better than no answer at all or the late exact answer.

14. SUBJECT TERMS

15. NUMBER OF PAGES
18

16 PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED

NSN 7540-01-280-5500

FINAL REPORT ON

MONOTONE APPROXIMATE QUERY PROCESSING

Contract No.: AFOSR-90-0193

Contract Period: April 1, 1990 — September 30, 1992

Principal Investigator

J. W. S. Liu 217 333-0135 janeliu@cs.uiuc.edu

Institution:

Department of Computer Science, University of Illinois

1 INTRODUCTION

This report summaries the work done under the contract No. AFOSR-90-0193. The contract period is from April 1, 1990 to September 30, 1992.

This project is concerned with means to provide approximate answers to queries. We have developed a monotone query processing scheme. (Publications on this scheme and its theoretical foundation are listed in Section 4.) This scheme allows a database system to provide for each query a series of intermediate answers that are approximations of the exact answer. The approximate answers improve monotonically in accuracy as more data are retrieved and processed to answer the query. If for any reason query processing must be terminated prematurely before the exact answer is produced, the latest intermediate answer, that is, the best approximate answer produced so far, is made available to the application. For many time-critical applications, a timely approximate answer that is sufficiently good is better than no answer at all or the late exact answer.

The underlying principle of the monotone query processing scheme is the approximate relational model. This model was initially developed as a rigorous framework for producing approximate answers to set-valued queries. A set-valued query has as its exact answer a set of objects with properties given by the query qualifications; in the relational model, such an answer is a relation. A meaningful and useful set of approximate answers can be defined in terms of all the subsets and supersets of the exact answer. The approximate relational model formally captures this semantics of approximation. In particular, this model defines the approximations of any standard relation in terms of supersets and subsets of the relation, a partial-order relation over the set of all approximate relations for comparing them, and a complete set of new relational algebra operations on approximate operands. Every one of these relational algebra operations is monotone in the sense that the result of the operation is better when its operand(s) becomes better. This model has since been extended to give meaningful approximation semantics for many frequently encountered types of single-valued queries.

We have implemented a prototype query processor, called APPROXIMATE, to demonstrate the feasibility of monotone approximate query processing. APPROXIMATE accepts standard relational algebra queries. It assumes that the data is stored as relations and the user's view remains that of the relational model. It can be implemented on a relational database system requiring little or no change to the underlying relational architecture. The query processor itself takes an object-oriented approach. Its views of the stored data are object-oriented. Such views provide it with the needed semantic support that is lacking in the relational model and enable it to keep the additional overhead in producing approximate answers small.

Following this introduction, Section 2 discusses the need to make query processing in real-time, dependable databases incremental and monotone. Section 3 summarizes the technical accomplishments of this project. Section 4 lists the publications on work supported by contract No. AFOSR-90-0193.

2. MOTIVATION AND OBJECTIVES

The objectives of this research is to develop the basic concept of incremental, monotone query processing in real-time, dependable database systems and to implement a prototype query processor to demonstrate this concept. In many (hard) real-time applications, such as machine vision, multiple robots,

and air traffic control, processes must share data stored and maintained by a database system. The work on incremental, monotone query processing was motivated by the need for database systems suited for these applications. It is often difficult to meet two requirements of real-time, highly-dependable database systems: satisfying timing constraints of time-critical query and update operations and providing fault tolerance and graceful degradation in the presence of host and network faults.

The *imprecise computation technique* has been proposed in 1987 as a way to make meeting these requirements easier. (Selected publications on this technique are [1-7].) Since its advent, the results in imprecise-task scheduling have clearly demonstrated the feasibility and effectiveness of this technique. We call a system based on this technique an *imprecise system*. In an imprecise system, each critical task is structured so that it is monotone. The intermediate results of increasing accuracy produced by each task as it executes are stored. If the task completes, the result produced by the task is the exact, desired one. If execution is not completed for any reason, then the latest intermediate result stored before the task terminates, that is, an *approximate result*, is made available. The portion of each task that must be completed for the approximate result produced by it to be sufficiently good and, hence, usable is considered to be *mandatory*. The portion of the task that refines the usable approximate result to further reduce the error in the result is considered to be *optional*. The imprecise computation technique eases the difficulty in meeting all deadlines at all times significantly for the following reason. To ensure that all deadlines are met, we only need to ensure that the mandatory portion of all critical tasks are completed by their deadlines. We can use the system resources left over after all mandatory subtasks are completed to complete the optional portions of as many tasks as possible. Only mandatory portions are restricted to have bounded execution time and resource requirements. It is not necessary to eliminate non-determinism in the timing and resource requirements of optional portions. This technique can also be used in a natural way to enhance the dependability of computing systems. A fault may cause a task to terminate prematurely. If the task has already produced a usable imprecise result when it terminates, no error-recovery action needs to be taken. If the imprecise result is not usable, the result, together with the recorded values of the accuracy indicator variables, gives a snapshot of the state of the computation at the time of its termination. This snapshot can serve as a checkpoint from where the computation can continue after the fault is cleared.

The imprecise computation technique relies on the use of well-behaved computational algorithms that produce acceptable, intermediate results in predictable, short amounts of time and better results when allowed more time. Until recently, such algorithms did not exist in the database domain. Traditionally, database query and update operations are atomic. An answer to a query is returned only after all the data required to answer the query are retrieved and processed. If a failure causes some of these data to become unavailable, no answer is returned. Atomicity of query processing operation is desirable for traditional database applications since one needs and, hence, is willing wait for exact answers. In contrast, a late time-critical answer from a real-time database may be less accurate than a sufficiently good and timely approximate answer. Time-critical data stored in a database deteriorates with time as the real-world it models changes. To support imprecise computations in the database domain, query computations must be restructured to allow for their partial, approximate completion.

Making query processing incremental and monotone is also a way to make database systems more fault tolerant. A host or network fault may make some of the data required to answer queries inaccessible

and cause query computations to terminate prematurely. By making an approximate answer available whenever an exact answer cannot be obtained, we can increase the availability data.

3. TECHNICAL ACCOMPLISHMENTS

We have built a prototype query processor, called APPROXIMATE. This query processor makes approximate answers to database queries available if part of the database is unavailable or if there is not enough time to produce an exact answer. In particular, the processor implements monotone query processing; the approximate answer returned by the processor improves as more data is retrieved to answer the query. As shown in Figure 1, APPROXIMATE accepts standard relational algebra queries. The data is stored as relations and the user's view remains that of the relational model. An object-oriented approach is taken to implement the query processor. The semantic information provided by the object-oriented view maintained by APPROXIMATE enables it to produce good initial approximations and to reduce the overhead in query processing. This processor can be implemented on a relational database system, requiring little or no change to the underlying relational architecture. This section describes its underlying data model, approximation semantics, query processing primitives and semantic support, monotone query processing algorithm and its implementation.

3.1. Approximation Semantics

Database queries can be divided into two types: set-valued and single-valued queries. The qualifications of a *set-valued query* define a property (or properties) of objects, and the expected answer is the set of objects with this property. A *single-valued query* expresses some properties of the object(s) of interest as a whole, and the expected answer may assume any of a countable set of values. We have developed meaningful approximation semantics of answers to these two types of queries.

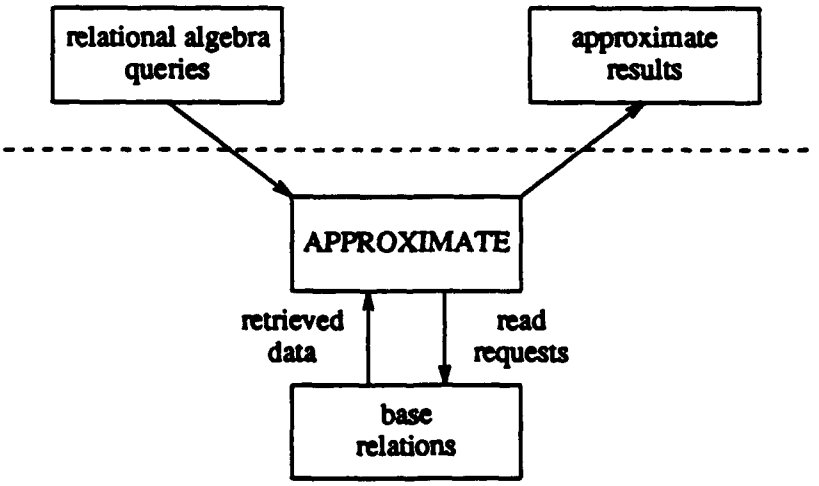

Figure 1. APPROXIMATE: A Monotone Query Processor

3.1.1 Approximate Relational Model

An exact answer E to any set-valued query is a set of data objects. In the traditional relational model, it is a (standard) relation. Meaningful approximate answers of such a query can be define in terms of subsets and supersets of the exact answer. This approximation semantics was suggested by Buneman, Davidson and Watters in [8] and is the basis of our approximate relational model. The approximate relational model defines for every standard relation, a set of approximate relations, a partial order relation over this set for comparing the approximate relations, and a complete set of approximate relational algebra operations for processing the approximate relations.

Specifically, a meaningful approximation of a set-valued exact answer E can be defined in terms of a subset and a superset of E. An approximate relation A of an exact answer E is composed of two sets of tuples: a *certain set* C, which is a subset of E, and a *possible set* P, where $A = C \cup P$ is a superset of E. This approximate relation is denoted by the 2-tuple (C, P). We refer to tuples in the certain set and possible set as *certain* tuples and *possible* tuples, respectively. (We sometimes also refer to tuples stored in the database as *database objects*.)

Any exact answer E has many approximate relations. Given a set of approximate relations of E, we defined a partial order relation $\geq$ over the set for comparing them as follows. One approximate relation is $A_i = (C_i, P_i)$ is equal to or better than another $A_j = (C_j, P_j)$, denoted as $A_i \geq A_j$, if $A_i \subseteq A_j$ and $C_i \supseteq C_j$. In other words, A_i is better than A_j because of the fact that the tuples in $C_i - C_j$ are in E becomes certain and/or the fact that the tuples in $A_j - A_i$ are not in E becomes certain. This partially ordered set of all approximate relations of E is a lattice. In the lattice, $A_0 = (\varnothing, \upsilon)$ is the least element and the worst possible approximation of E, where $\varnothing$ denotes the null set and υ is the cartesian product of all the domains in the schema of E, the set of all possible tuples which could be in E. A key assumption made in APPROXIMATE is that the number of elements in υ is finite and, at least theoretically, can be computed without accessing any data. The greatest element of the lattice is the best possible approximation of E and is E itself, which is represented by $(E, \varnothing)$.

The example shown in Figure 2 illustrates that the semantics of approximation defined above matches our common sense notion. Suppose that an AIRPORTS database that resides on-board an airplane. AIRPORTS contains the relation RUNWAYS (id, airport, length, obstructions). A pilot queries the database to locate all the runways with an easterly direction, with ids from 5 to 13, at airports O'Hare (ORD) and Midway (MDW) in Chicago. The exact answer to this query is the relation E shown in Figure 2(a). Figures 2(b)-(f) show five other relations. The relation R_1 shown in Figure 2(b) contains tuples on all runways at ORD and MDW. It gives all the tuples that are possibly in E and hence, is an approximation of E. The relation R_2 shown in Figure 2(c) contains tuples on all easterly runways at ORD airport. It is a subset of E, containing tuples that are certainly in S. By the definition above, R_2 is not an approximation of the E. R_3 shown in Figure 2(d) is another approximation of E. The first two tuples are certainly in E. They form a subset of E. The last four tuples are possibly in E. R_3 is a superset of E. It is a subset of R_1 and is a better approximation of E than R_1. The relation R_4 in Figure 2(e), another superset of E, is the set of tuples on easterly runways in Illinois. It is a different superset of E than R_1 and is not comparable to R_1. The approximate relation R_5 in Figure 2(f) is a subset of R_4 and a superset of E. R_5 is better than R_4, but is not comparable to R_1 and R_3.

id	Airport	Length
6	MDW	7500
7	MDW	8000
9	ORD	11000
10	ORD	8000
13	ORD	10000

(a). E: All easterly runways at ORD and MDW

id	Airport	Length
1	MDW	6000
6	MDW	7500
7	MDW	8000
9	ORD	11000
10	ORD	8000
13	ORD	10000
27	ORD	9500

(b). R_1: All runways at ORD and MDW

id	Airport	Length
9	ORD	11000
10	ORD	8000
13	ORD	10000

(c). R_2: Easterly runways at ORD

id	Airport	Length
6	MDW	7500
7	MDW	8000
9	ORD	11000
10	ORD	8000
13	ORD	10000
27	ORD	9500

(d). R_3: An approximation of E

id	Airport	Length
5	CMI	10000
6	MDW	7500
7	MDW	8000
8	PIA	8500
9	ORD	11000
10	ORD	8000
13	ORD	10000
13	SPI	11500

(e). R_4: All easterly runways at Illinois airports

id	Airport	Length
6	MDW	7500
7	MDW	8000
10	ORD	8000
5	CMI	10000
9	ORD	11000
13	ORD	10000
13	SPI	11500

(f). R_5: An approximation of E

Figure 2. An Example: Approximate Relations

3.1.2 Approximations of Single-Valued Answers

The semantics of approximation defined by the approximate relational model can also serve as a basis for meaningful semantics of approximation of many types of single-valued queries. An exact answer to a single-valued query can be a single object retrieved from the database, the value of an aggregate function (such as "count"), or a value (such as "yes" or "no"), derived from a set-valued or single-valued answer. Examples of single-valued queries include:

(1) "In what city is airport MDW?"

(2) "What is the maximum length of the runways at O'Hare?"
(3) "How many pints of type O- blood are available?"
(4) "Is the number of runways at O'Hare greater than 10?"
(5) "What is the color of car No. 20?"

Query (1) exemplifies a special case of set-valued queries whose exact answer is a set of cardinality 1. The exact answer E is "Chicago". Since any proper subset of E is the null set $\varnothing$, an approximate answer of E is, therefore, simply $(\varnothing, P_1)$ where the possible set P_1 is a superset containing the exact answer. A possible value of P_1 is {Chicago, Peoria, Champaign}, a set of three cities near Chicago. This answer improves as elements "Peoria" and/or "Champaign" are deleted from P_1. We can consider query (2) to be another set-valued query whose exact answer contains one element. The exact answer of 10,000 ft is an element of a set P_2 of possible values, such as the set P_2 = {8000, 9000, 10000, 11000} of known runway lengths. Therefore, approximate answers in this case are also of the form $(\varnothing, P_2)$ for all supersets of {10000}.

Equivalently, we can consider the exact answer of query (2) as the value of the aggregate function "max" over the set L of lengths of all runways at O'Hare. A meaningful approximate answer in this case is again a range of values that contains the exact answer. Upper and lower limits of such a range are the values of the function "max" over a superset and a subset of the set L, respectively. Similarly, the exact answer to query (3) is the value of an aggregate function over the set O of all objects on type O- blood. In this case, the aggregate function is "count" or "sum." Since the domain of the sum function is a set of non-negative numbers in this case, as an approximation to the value of such an aggregate function, we can provide the values of the aggregate function defined over the certain set $C \subseteq O$ and the superset $C \cup P$ where (C, P) is an approximation of O. Again, these values give us a range of values, such as "at least 2", or "2 to 10".

Query (4) requires a yes/no answer. Such an answer is derived from the aggregate function "count" over the set of data objects that satisfy the query qualifications. Specifically, the exact answer of query (4) is the value of a binary function of the value of "count" over the set R of all runways at O'Hare. The derived exact answer is "no" whenever the value of the count function is equal to or less than the threshold value 10. Again, a meaningful approximation of the exact answer can be derived from an approximation (C, P) of the set R. In particular, as long as the certain set in (C, P) contains 10 elements or less, the derived approximate answer remains "no". In addition to this value, we can also provide, as part of an approximate answer to such a query, a percentage value and the value of count over the current certain set C. An example is "no - 60%, number of runways ≥ 3". The percentage value, called the *confidence factor*, in the approximate answer gives the amount of data retrieved and processed thus far to produce (C, P). The value of the count function over the certain set (3 in the above example) gives a lower bound on the exact value of the count function. As more data are retrieved and processed, the confidence factor increases monotonically, and the value of the count function over C becomes closer to the exact value. In this sense, the approximate answer monotonically improves.

We note that in the cases of queries (1) - (4) there is a natural partial order between the elements of the attribute domain over which approximate answers are defined or are derived from. In query (2), for example, the exact answer and approximate answers are expressed in terms of positive integers; there is a unique way to compare different runway lengths. In query (1), the exact answer is an element of the set

of geographical locations. It is natural to order the closeness of different geographical locations to the exact location by physical distances to it. Query (5) illustrates that the exact answer may be an element or derived from elements of an attribute domain whose elements can be compared according to more than one partial-order relation. Suppose that the exact answer E is the color "maroon". A superset of colors containing "maroon" is an approximate answer. The elements of the color domain can be compared according to different criteria based on different characteristics of the color attribute. For example, we can identify a superset of colors close to maroon based on the color spectrum, or a superset of colors close to maroon based on color intensity. Hence, a superset of E can be the set of all reddish colors, such as {red, pink, maroon}, or a superset of all dark colors, such as {navy, maroon, black}. APPROXIMATE defines and supports by default some alternative partial-order relations to demonstrate the feasibility of providing different approximate answers, but does not provide the needed user interface through which a user can define other partial-order relations, and hence new semantics and comparison criteria, on a per query basis.

3.2. Distance Functions

Thus far, we have discussed how subsets of a domain are compared solely on the basis of their cardinalities. In query (1), for example, the approximate answers (∅, {Champaign, Chicago, Paris IL}) and (∅, {Champaign, Chicago, Paris France}) are either not comparable or are equally good initial approximations of ({Chicago},∅). We can make the approximation semantics more meaningful by comparing subsets not only on the basis of their cardinalities, but also on the basis of some metric that measures the distances of their elements to the exact answer. In particular, we use a metric of accuracy, called a *distance function*, to quantify how much better one approximate answer is than another. A distance function induces a partial-order relation to be defined over the set of all approximate answers of an exact answer E. This partial-order relation is a lattice in which the unique best element is the exact answer itself. The query processor uses this measure of accuracy to identify a good initial approximation of the exact answer when query processing starts. It then chooses data to retrieve and process so that it produces a series of approximate answers of improving accuracy according to the relation. This series converges to the exact answer when query processing terminates normally.

3.2.1. Distance Between Elements and Vector Distances

Using an approach similar to VAGUE [9], we define one or more distance functions for comparing elements of an attribute domain. There can be more than one distance function for a domain. The distance between two values of an attribute, that is, two elements of the domain of the attribute, is defined as follows. Given an attribute domain D, a distance function for D is a mapping δ from the cartesian product $D \times D$ to the set of non-negative reals that is

reflexive: $\delta(v_i, v_i) = 0$, for every value v_i in D,
symmetric: $\delta(v_i, v_j) = \delta(v_j, v_i)$, for all values v_i and v_j in D, and
transitive: $\delta(v_i, v_j) + \delta(v_j, v_k) \leq \delta(v_i, v_k)$, for all values v_i, v_j and v_k in D.

If one value v_i cannot be used to approximate another value v_j, then the values are not comparable, and we denote this by $v_i \infty v_j$. For any two pairs of values, (v_i, v_k) and (v_j, v_k), the value v_i is said to be closer to v_k than v_j, or v_i is better than v_j, if $\delta(v_i, v_k) < \delta(v_j, v_k)$. Figure 3 illustrates an example of the distances between several values of the attribute color. According to the distances based on color

spectrum, we say that maroon is closer to red than green, since δ(red, maroon) = 1 < δ(green, maroon) = 4.

The qualifications of a query can involve many attribute values. To provide a measure of quality in this case, we define the distance between pairs of values of two attributes as follows. Given m attributes and their domains $D_1, D_2 \cdots D_m$, a vector value over $\mathbf{D} = (D_1 \times D_2 \times \cdots \times D_m)$ is denoted by $v_i = (v_{i1}, v_{i2} \cdots v_{im})$ where $v_{i1} \in D_1, v_{i2} \in D_2 \cdots v_{im} \in D_m$. An m-dimensional distance function is the mapping from $\mathbf{D} \times \mathbf{D}$ to the set of m-tuples of non-negative reals that is reflexive, symmetric and transitive.

Elements of a set V of m-dimensional vector values can be used to approximate a particular vector v in this set. By giving a vector distance function δ different semantics, we can define either a linear-order or a partial-order relation over the set of vector values. For example, given two 2-dimensional vectors v_i and v_j with $\delta(v_i, v) = (d_1, d_2)$ and $\delta(v_j, v) = (d_1', d_2')$, we can choose to compare v_j and v_i as approximations of v according to the distances between the values of the individual attributes in a lexicographical order. This interpretation of the 2-dimensional distance function gives us a linear-order relation over the set V. Alternatively, a linear-order relation can be defined by assigning a weight w_i to each dimension in the vectors. The weighted distances of the individual dimensions are added to produce a single distance value. We can say that v_j is better ($v_j \geq v_i$) if $(w_1 d_1' + w_2 d_2') \leq (w_1 d_1 + w_2 d_2)$, otherwise $v_i \geq v_j$. We can also define a partial-order relation over V as follows: for any two 2-dimensional vectors v_i and v_j whose distances to v are $\delta(v_i, v) = (d_1, d_2)$ and $\delta(v_j, v) = (d_1', d_2')$, $v_j \geq v_i$ if $d_1' \leq d_1$ and $d_2' \leq d_2$, that is $\mathbf{d}' \leq \mathbf{d}$; otherwise the values are not comparable, denoted as $v_i \infty v_j$. This defines a partial order relation that is reflexive, symmetric, and transitive. Linear-order and partial-order relations over m-dimensional values can be defined in an analogous way as for 2-dimensional values.

Query (5) mentioned earlier illustrates the fact that we sometimes may want to define a vector distance function over a single attribute domain. An m-dimensional distance function is a mapping δ from the set $D \times D$ to the set of m-tuples of non-negative reals that is reflexive, symmetric and transitive. For example, we can define a 2-dimensional distance function on the domain of colors. This distance function allows us to compare colors according to two characteristics: color spectrum and color intensity.

color	*color*	Distance
red	maroon	1
red	orange	1
red	green	3
maroon	orange	2
maroon	green	4
orange	green	2

(a). Distances based on color spectrum

color	*color*	Distance
red	maroon	2
red	orange	2
red	green	1
maroon	orange	1
maroon	green	2
orange	green	2

(b). Distances based on color intensity

Figure 3. An example of distance functions

In Figure 3 the 2-dimensional vector distance between the color red and the color green is δ(red, green) = (3,1). The distance between red and green based on characteristic of color spectrum is 3, and the distance between red and green based on the characteristic of color intensity is 1. Depending on the interpretation of the 2-dimensional distance function, a linear or partial-order relation over the domain of colors can be defined.

3.2.2. Distances between Subsets

Let $V = \{v_1, v_2 \cdots v_m\}$ be a nonempty subset of D. We can define the distance $\delta(v, V)$ between a value v in V and the subset V as the average or maximum of the distances between the values in V and the value v. A value v_i in D is not comparable to a subset V of D, denoted by $v_i \infty V$, if v_i is not in V. Such a function δ defines a linear-order relation over V. v_i is closer to V than v_j if $\delta(v_i, V) < \delta(v_j, V)$. For example, according to the distance function defined as the averages of the distances in Figure 3, the color red is closer to the given subset {maroon, green, orange, red} than the color green because δ(red, {maroon, green, orange, red}) = 1.25 and δ(green, {maroon, green, orange, red}) = 2.25. We also define the distances between a value and a set of values of m different attributes as follows. Given a vector of values $\mathbf{v} = (v_1, v_2, \cdots v_m)$ and an m-dimensional vector $\mathbf{V} = (V_1, V_2, \cdots V_m)$ of sets where $V_1 \subseteq D_1$. $V_2 \subseteq D_2 \cdots$ and $V_m \subseteq D_m$, the value of the distance function $\delta(\mathbf{v}, \mathbf{V})$ is the vector $\mathbf{d} = (d_1, d_2, \cdots d_m)$, where $d_1 = \delta(v_1, V_1), d_2 = \delta(v_2, V_2), \cdots, d_m = \delta(v_m, V_m)$.

The distance $\delta(v, V)$ between a value v and a subset V containing v also allows us to measure the goodness of V as an approximation of v. To ensure that the approximate answers obtained at different stages of query processing are consistent, two subsets V_i and V_j of D are comparable only if either $V_i \subseteq V_j$ or $V_j \subseteq V_i$. By *consistency* here, we mean that a better approximate answer obtained later as more data is retrieved and processed never contradicts a poorer approximate answer obtained earlier. Given two comparable subsets V_i and V_j, we say that V_i is a better approximation of V_j than v if $\delta(v, V_i) < \delta(v, V_j)$. In the example in Figure 3, the subsets {maroon, orange, red} and {maroon, green red} are not comparable. The query processor never produces one of these answers earlier and the other one later. On the other hand, {maroon, green, red} is a better approximation of red than {green, red} because δ(red, {maroon, green, red}) = 1.33 and δ(red, {green, red}) = 1.5. However, {maroon, red} is better than {maroon, green, red}.

The semantics of approximation given by the approximate relational model discussed earlier can be defined in a similar manner in terms of the following distance function over all the subsets of an attribute domain D (or the cartesian product of m attribute domains). According to this semantics of approximation, two approximate answers $A_i = (C_i, P_i)$ and $A_j = (C_j, P_j)$ of a set-valued answer E can be compared only if either $A_i \subseteq A_j$ and $C_i \supseteq C_j$ or $A_j \subseteq A_i$ and $C_j \supseteq C_i$. Specifically, comparable subsets (or supersets) of the exact answer are compared solely on the basis of their cardinalities. The distance between an approximate answer $A = (C, P)$ to the exact answer E of a set-valued query is the 2-dimensional vector:

$$\delta(A, E) = (|E - C|, |C \cup P - E|) = (|E - C|, |P - E|) = (d_1, d_2).$$

In this expression, d_1 is the distance between the certain part and the exact answer and is equal to the number of objects that are in E but are not in the certain set C. The d_2 is the distance between the approximate answer $A = (C \cup P)$ and the exact answer and is equal to the number of objects in the

possible set P that are not in E. In the example in Figure 2, the distance between R_1 in Figure 2(b) and the exact answer E in Figure 2(a) is $\delta(R_1, E) = (5,2)$. The distance between R_3 in Figure 2(d) and E is $\delta(R_3, E) = (3,1)$. The partial-order relation defined is the same as the one defined by the approximate relational model, but the distance between an approximate answer to an exact answer gives us more information on the sizes of the subsets and supersets used to form the approximate answer.

3.3. Monotone Approximate Operations and Query Processing Algorithm

APPROXIMATE uses as primitives a set of approximate relational algebra operations, because standard relational algebra operations cannot operation on approximate relations. These approximate relational algebra operations are defined in Table 1. Each operation accepts an approximate relation(s) as an operand and produces an approximate relation as its result. Specifically, Table 1 defines approximate union, set difference, select, project, and cartesian product. Join can be derived from select and cartesian product, and intersection from set difference. In the definitions, $\times$, $\cup$, $\cap$, and - in the C_T and P_T columns denote the set-theoretical operations cartesian product, union, intersection and set difference, respectively. The approximate operations have the *monotonicity* property. Here, we say that an operation is *monotone* if the result of the operation is better when its operands are better.

To explain the monotone query processing algorithm implemented in APPROXIMATE, we first note that any relational algebra query can be represented as a query tree. Each leaf node of this tree represents a ***base relation*** to be read by the query processor from the database. Each non-leaf node represents the result of a relational algebra operation. The root node represents the final result of the query. Query answers are traditionally derived in the following all-or-nothing manner. Each leaf node is evaluated by issuing a read request to the database for the base relation it represents. A node at the next higher level can be evaluated only when all operands represented by its children are available. The query evaluation process ends when the root node is evaluated. The value given to the root node is the exact answer to the query. In contrast, APPROXIMATE processes each query in a monotonic, incremental

Approximate Operation	C_T	P_T
Union: $R_T = R_1 \cup R_2$	$C_T = C_1 \cup C_2$	$P_T = (P_1 \cup P_2) - C_T$
Difference: $R_T = R_1 - R_2$	$C_T = C_1 - R_2$	$P_T = (P_1 - R_2) \cup (P_2 \cap R_1)$
Select: $R_T = \sigma_{att=val} R_1$	$C_T = \sigma_{att=val} C_1$	$P_T = \sigma_{att=val} P_1$
Project: $R_T = \pi_{att} R_1$	$C_T = \pi_{att} C_1$	$P_T = \pi_{att} P_1$
Cart. Prod: $R_T = R_1 \times R_2$	$C_T = C_1 \times C_2$	$P_T = (R_1 \times R_2) - C_T$

Table 1: Approximate relational operations

manner. Each node in the query tree represents an approximate relation. An additional primitive is the *approximate read_request* which we use instead of a standard read_request. Each approximate read_request returns a segment of the requested base relation at a time. An approximate read_request can be implemented easily whenever a stored relation is horizontally partitioned across a file system or a relation is indexed. Otherwise, when relations are not partitioned or indexed, an approximate read_request is the same as a standard read_request.

In principle, the basic monotone query processing algorithm works as follows. It begins by assigning an initial approximation to each node in the query tree. Such an initial value, theoretically, can be the cartesian product of all domains in the schema of the relation represented by the node. As each approximate read_request is carried out, each returned segment causes additional certain tuples to be added to, and possible tuples to be deleted from, the current approximation of the base relation. The value of the leaf node improves as more segments are returned. The improvement in the leaf nodes is propagated upward to the root node by reevaluating the nodes in the query tree. The value of the root node is updated with better values each time the root node is reevaluated. Thus APPROXIMATE is capable of producing a chain of increasingly better approximate answers, each integrating the effect of additional base relation data. None but the final, exact answer requires all base relation data be available before it can be produced. If query processing terminates prematurely, some approximate answer in the chain will be returned and the quality of the returned answer increases monotonically with time.

3.4. Implementation of Monotone Query Processing

To implement monotone query processing efficient, APPROXIMATE maintains semantic information about the stored data so that initial approximations of different answers can be chosen and generated. The rate at which the approximate answers converge to the exact answer can be made faster by choosing better initial approximations. We must also implement the approximate relational algebra operations efficiently. Every approximate relational algebra operation defined in Table 1 involves operations on the possible tuples as well as the certain tuples in its operand(s). Substantial query processing time is saved in APPROXIMATE by avoiding operations on the possible tuples as much as possible. This section describes the object-oriented view maintained by APPROXIMATE to provide it with the needed semantic support and the strategy used to defer and avoid operations on possible tuples.

3.4.1. Object-Oriented View

APPROXIMATE views a base relation, and sometimes a segment of the relation, as a class; tuples in the relation, or the segment, are instances of the corresponding class. Specifically, in the APPROXIMATE's view, data objects with common characteristics are organized into classes. Classes are in turn organized into a collection of class hierarchies. The definition of each class provides a semantic interpretation of the corresponding segment of the relation, and each class hierarchy provides relationships between classes. APPROXIMATE may use more than one class hierarchy to give different views of the data objects in each relation. APPROXIMATE makes use of these relationships between classes and the values of a default distance function when choosing initial approximations. It stores its class hierarchies in memory for fast access. Information supplied by a class hierarchy is accessed along with the base relations during query processing.

In addition to a semantic interpretation of the database objects, a hierarchy of classes also provides access path information. A type of access path information is the granularity of stored data for retrieval at a time. Some base relations are returned only in their entirety. An approximate read_request issued against such a relation returns the entire relation. On the other hand, individual segments of other base relations can be retrieved one at a time. This information is provided by every class hierarchy. In particular, each class at the leaf level of a class hierarchy represents the smallest unit of stored data that can be returned at one time by an approximate read_request. The segments represented by the leaf subclasses of a class can be returned one at a time by multiple approximate read_requests to retrieve the individual segments or by a single approximate read_request to retrieve all the segments together as one unit.

Each class in a class hierarchy has a class variable D called a *domain range*. The domain range of a class gives the ranges of values that the attributes of the instances of the class can have. A relation, or a segment of it, on the domains $D_1, D_2, \cdots D_m$ is a subset of $D_1 \times D_2 \times \cdots \times D_m$. In the class corresponding to the relation (or a segment of the relation), the value of the domain range D is the m-dimensional vector $(V_1, V_2, \cdots V_m)$ where, for $i=1, 2, \cdots, m$, V_i is the subset of D_i containing all possible values of the ith attribute of the instances of the class. When it is necessary to return an approximate answer whose possible set contains the set of all instances of the class, the values of the possible tuples can be computed from the values of this class variable.

3.4.2. Approximate Classes and Objects

During query processing, APPROXIMATE generates a template of the possible tuples P for the approximate relation at each node in the query tree. Rather than performing repeated relational algebra operations on the actual possible tuples every time an update of the value of a leaf node is propagated up the query tree, it modifies the templates. The template and the methods for modifying the template of possible tuples for the approximate relations of each standard relation are provided in an *approximate class*. An approximate class provides a description of the set of all approximate relations of a standard relation E. An approximate class has two instance variables: Certain_Part and Possible_Part. The domain of the variable Certain_Part is the class C, referred to as the *certain class*. The class C consists of the attributes of the tuples in the certain set of an approximate relation of E. Instances of C are the certain tuples in an approximate relation of E. The domain of the variable Possible_Part is a metaclass whose instances are themselves classes. We denote this set of instances that are classes by P. We refer to the classes in P as *possible classes*. The instances of a possible class are the possible tuples in an approximate relation of E. An approximate class also has a class variable OP that has as its value a relational algebra operator or approximate read_request. We will return to discuss this variable.

An instance of an approximate class is an *approximate object*. Each value of the approximate object corresponds to an approximate relation. A value $\mathbf{R}_i$ of an approximate object, corresponding to the approximate relation $R_i = (C_i, P_i)$, is described by the 2-tuple $(C_i, \mathbf{P}_i)$; tuples in the certain set C_i are instances of the certain class C, and $\mathbf{P}_i$ is a set of possible classes, the set of all instances of which is the possible set P_i. As an example, Figure 4 shows the value $\mathbf{R}_5$ of an approximate object corresponding to the relation R_5 in Figure 2(f). The set C_5 of certain tuples in $R_5 = (C_5, P_5)$ are a subset of the instances of the class "IL short E runways" of runways at O'Hare airport with no obstructions. The possible tuples

id	Airport	Length
6	MDW	7500
7	MDW	8000
10	ORD	8000
{IL-long-E-runways}		

Figure 4. R_5: An approximate object

in P_5 are instances of P_5 = {IL-long-E-runways}, the class of long, easterly runways in Illinois.

The approximate relational algebra operations in Table 1 are implemented in APPROXIMATE by their corresponding extended approximate relational algebra operations that operate on the corresponding approximate objects. These operations are the primitives in APPROXIMATE used for query processing. The extended operations make use of the information contained in the possible classes. The relational algebra operations are performed on the certain instances, as in traditional query processing. However, rather than manipulating possible tuples, the operations are done on the possible classes. The extended operations are monotone as are the corresponding approximate relational algebra operations they implement.

3.4.3. Processing of Relational Algebra Queries

After a query is parsed, APPROXIMATE creates an approximate class for each node in the query tree. This class gives a template of all approximate relations of the relation represented by the node. The value of the class variable OP of an approximate class at a non-leaf node is the relational algebra operation whose result is the value of the node. In the case of a leaf node, the value of OP is an approximate read_request. The schema of the relation represented by a node in the query tree can be determined from the schemas of the base relations in the subtree rooted at this node and the relational algebra operations at and below this node. This schema determines the attributes in the certain class C of the approximate class. APPROXIMATE makes use of the information on the schemas of all base relations provided by its view of the database to accomplish this task.

Selections of Initial Approximations and Improvement Propagation

An approximate object is then created for each approximate class. An approximate object is treated as a temporary object which only exists during the processing of the query. The initial value of an approximate object of each node in the query tree is determined by APPROXIMATE as follows. The set of instances of the certain class of every approximate object is initially empty. For the possible classes of an initial approximation at a leaf node, APPROXIMATE chooses, from a class hierarchy, the subclasses of the class that corresponds to the base relation represented by the node. The possible classes of a non-leaf node of the query tree are determined from the possible classes and the extended relational algebra operations at and below the node.

For some queries, APPROXIMATE can improve the initial approximations in the query tree. APPROXIMATE maintains a domain range table of attributes and the domain range values that are used to categorize the tuples into classes in its class hierarchies and to compute the distances between attribute values. If an attribute in a select operation on a base relation is an entry in the table, the most specialized class whose domain range contains the attribute value in the select operation is chosen as the possible class in the initial approximation of the leaf node. The segment corresponding to that class is retrieved when the leaf node is evaluated. An example is a query that contains "select the IL runways where length = 7500". Suppose that runway length is an attribute contained in the domain range table, and in the view about the RUNWAYS relation, the class "IL short runways" is the most specialized class whose domain range value of runway length is $\{0 - 9500\}$. This range contains, and has the smallest distance to, 7500 ft. This class is chosen to be the possible class in the initial approximation of the leaf node. Moreover, rather than reading the entire relation RUNWAYS, an approximate read_request to retrieve the objects that are instances of the class "IL short runways" is issued.

Efficient Implementation

An overwhelmingly large amount of overhead can incur if each of the operations is performed repeatedly on the same tuples that have been added to the certain set during previous updates and on the possible classes that remain members of the possible set. To avoid repeated operations on the same certain tuples, the processing of the certain set(s) is done incrementally. During each update, the operation at each node is applied only to the certain tuples that have migrated to the certain set of the operand in the most recent update. The resultant certain set produced by the operation is the union of the subset of certain tuples obtained in this update with the certain set produced during previous updates. Only the newest certain tuples are sent upwards in the query tree along with the possible classes. Specifically, let $\mathbf{R}_j$ and $\mathbf{R}_i$ be the values of the approximate object at a node before and after the latest update at the node. The certain tuples in $C_i - C_j$ and the set $\mathbf{P}_i$ of possible classes in $\mathbf{R}_i$ are sent to the approximate object at the parent node. At the parent node only the certain tuples in $C_i - C_j$ are processed. Therefore, the total work done in performing an approximate relational operation on increments of the certain set(s) until query processing terminates normally is the same as in traditional query processing. In other words, monotone approximate query processing does not increase the number of operations performed. The only exceptions are the difference and cartesian-product operations. We make use of the semantic information provided by the possible classes to make the difference operation more incremental. Similarly, some additional overhead (beyond the unavoidable amount also incurred in traditional query processing) can occur when the cartesian product is performed. Although the approximate cartesian product is applied to the same number of tuples as during the computation of the traditional cartesian product operation, because only a subset of the certain tuples is available at a time, we cannot use any optimizing method for the cartesian product which requires access to the complete set of tuples at a time to speed up this operation.

An additional cost in monotone approximate query processing over and above the cost in traditional query processing is the time spent to process the possible classes and to generate the possible tuples in the final answer. To minimize this cost, processing of the possible classes is deferred in APPROXIMATE to as late as possible. For the set of possible classes of each approximate object, APPROXIMATE maintains a *symbolic expression* of extended relational algebra operations and their operand(s); when this

symbolic expression is evaluated, it produces the values of the possible tuples in the object. The symbolic expressions of all approximate objects are created before any data is retrieved. A symbolic expression at each node is obtained by appending the operation of the node to the symbolic expression(s) of its operands(s). The symbolic expression of the root node is evaluated only when the user requests the evaluation or query processing terminates and an approximate answer is provided.

In summation, the overhead required to provide approximate answers above and beyond the overhead required in traditional query processing is that of identifying the initial approximation and including the possible classes in the query computation. In order to identify initial approximations, APPROXIMATE accesses the domain range table containing information about the attribute values and classes, as well as class and distance information from the view it maintains. APPROXIMATE makes one class hierarchy access per base relation or segment to be retrieved. The total number of accesses required to identify all initial approximations is $O(k + j)$ where k is the number of selection operations in the query and j is the number of segments to be retrieved by the approximate read_requests.

Most of the overhead for processing the possible classes is that of sending the symbolic expression with each update from a child node to a parent node in the query tree. Contrasted with the size of the set of certain tuples sent with each update, the size of a symbolic expression is small. It is $O(n + m)$ where n is the total number of operations and m is the maximum number of possible classes in the symbolic expression. The overhead required to evaluate a possible class in a symbolic expression is the time to apply relational algebra operations to the classes. Depending on the relational algebra operation, this time is either $O(1)$ or $O(m)$ where m is the number of attribute values in the domain range. Hence the total time required to evaluate all possible classes is $O(m \times n)$.

4. PUBLICATIONS AND SOFTWARE PROTOTYPE

This section lists publications on work supported by this contract and describes the prototype query processor APPROXIMATE.

4.1. APPROXIMATE

APPROXIMATE is a query processor that produces approximate answers to relational algebra queries. The approximate answers have non-decreasing accuracy as more data are retrieved and processed to produce them. The database is assumed to be complete and the query precise. The imprecision in an answer occurs because time constraints or failures prevent all the data required to produce the answer from being retrieved and processed. When all the data are retrieved and processed, the final answer is the exact answer.

The prototype query processor displays the approximate answers as they are produced during processing. Its interactive interface allows the user to stop the processor when an approximate answer produced is good enough and to request query processing be continued if the approximate answer produced thus far is not yet useful. The current version of APPROXIMATE is implemented in Smalltalk, and we plan to implement an improved version in C++. The query processor can be interfaced to a relational database system with little or no change required to the underlying relational architecture. Its relies on an object-oriented view of the database for semantic support. The efficiency of the approximate query processor was increased by using a lazy evaluation of its operations, and approximate answers are

produced in a predictable amount of time. This work was partially supported by the U.S. Navy ONR Contract No. NOOO14-89-J-1181 and NASA Contract No. NAG 1-613.

4.2. Journal and Conference Publications

(1) Vrbsky, S. and J. W. S. Liu, "An Object-Oriented Query Processor That Returns Monotonically Improving Answers," *Proceedings of 1991 IEEE International Conference on Data Engineering*, pp. 472-481, Kobe, Japan, April 1991.

(2) X. Song and J. W. S. Liu, "How Well Can Data Temporal Consistency Be Maintained," *Proceedings of IEEE Symposium on Computer-Aided Control System Design*, Napa, California, March 1992.

(3) Vrbsky, S. and J. W. S. Liu, "Producing Approximate Answers to Database Queries," to appear in *Proceedings of SOAR'92 Conference*, Houston, TX, August 1992.

(4) Zhao, W., S. Vrbsky, and J. W. S. Liu "Imprecise Scheduling in Multiprocessor Systems," *Proceedings of the 5th International Conference on Parallel and Distributed Computing and Systems*, Pittsburgh, PA, September 1992.

(5) Vrbsky, S. and J. W. S. Liu, "Producing Approximate Answers to Set-Valued and Single-Valued Queries with APPROXIMATE," *Proceedings of CIKM-92 International Conference on Information and Knowledge Management*, pp. 405-412, Baltimore, Maryland, November 1992.

(6) Vrbsky, S. and J. W. S. Liu, "Producing Monotonically Improving Approximate Answers to Database Queries," *Proceedings of IEEE Workshop on Imprecise and Approximate Computation*, Phoenix, Arizona, pp. 72-75, December 1992.

(7) Vrbsky, S, "APPROXIMATE, A Query Processor That Produces Monotonically Improving Approximate Answers," Ph.D. Thesis, Department of Computer Science, University of Illinois, January 1993; to appear as a technical report.

(8) Vrbsky, S. and J. W. S. Liu, "An Object-Oriented Query Processor for Returning Monotonically Improving Partial Answers," to appear in *IEEE Transactions on Knowledge and Data Engineering*.

REFERENCES

[1] Liu, J. W. S., S. Natarajan, and K. J. Lin, "Scheduling Real-Time, Periodic Jobs Using Imprecise Results," *Proceedings of Eighth Real-Time Systems Symposium*, pp. 252-260, San Jose, CA, December 1987.

[2] Chung, J. Y., J. W. S. Liu, and K. J. Lin, "Scheduling Periodic Jobs That Allow Imprecise Results," *IEEE Transactions on Computer*, Vol. 39, No. 9, pp. 1156-1174, September 1990.

[3] Liu, J. W. S., K. J. Lin, W. K. Shih, A. C. Yu, J. Y. Chung, and W. Zhao, "Algorithms for Scheduling Imprecise Computations," *IEEE Computer*, Special Issue on Real-Time Systems, May 1991.

[4] Shih, W. K., J. W. S. Liu and J. Y. Chung, "Algorithms for Scheduling Tasks to Minimize Total Error," *SIAM Journal of Computing*, July 1991.

[5] K. J. Lin, J. W. S. Liu and K. Kenny, "FLEX: A Language for Programming Flexible Real-Time Systems," pp. 251-290, in *Foundations of Real-Time Computing: Scheduling and Resource Management*, Edited by A. M. Van Tilborg and G. M. Koob, Kluwer Academic Publishers, 1991.

[6] Liu, J. W. S., K. J. Lin, C.L. Liu, and C. W. Gear, "Imprecise Computation," in *Mission Critical Operating Systems*, A. K. Agrawala, et al ed., IOS Press, 1992.

[7] Liu, J. W. S., K. J. Lin, W. K. Shih, and J. Y. Chung, "Imprecise Computation: a Means to Provide Scheduling Flexibility and Enhance Dependability," in *Readings on Hard Real-Time Systems*, IEEE Press.

[8] Buneman, P., S. Davidson, and A. Watters, "A Semantics for Complex Objects and Approximate Queries," *Proceedings of the 7th Symposium on the Principles of database Systems*, pp. 305-314, March 1988.

[9] Motro, Amihai, "VAGUE: A User Interface to Relational Databases that Permits Vague Queries," *ACM Transactions on Office Information Systems*, Vol. 6, No. 3, pp. 187-214, July 1988.